UNIT 2

GREEDY ALGORITHMS

Patricia Cuesta Ruiz

Isabel Blanco Martínez

EXERCISE 2

A set of n files must be stored on a magnetic tape (sequential path storage medium), each file having a known length l1, l2, …, ln. To simplify the problem, it can be assumed that the reading speed is constant, as well as the information density in the tape.

The rate of use of each stored file is known in advance, that is, the number of pi requests corresponding to the file i that will be made is known. Therefore, the total of requests to the support will be the quantity $\boldsymbol{P = \sum_{i=1}^{n} p_i}$. After requesting a file, when it is found, the tape is automatically rewound until its beginning.

The objective is to decide the order in which the n files should be stored so that the average loading time is minimized, creating a correct greedy algorithm.

Code:

```python
class files(object):
    length = 0
    requests = 0
    def __init__(self, x, y):
        self.length = x
        self.requests = y

def initialize_tape(n):
    """
    OBJ: initialize the vector with n files and random values
    """
    from  random import randint
    files_list = []
    for _ in range(n):
        files_list.append(files(randint(1, 50), randint(1, 15)))
    return files_list

def order_tape(f):
    """
    OBJ: order the magnetic tape using bubble sort by dividing the length by the request number
    """
    for i in range(len(f)):
        for j in range(0, len(f)-i-1):
            if ((f[j].length / f[j].requests) > (f[j+1].length / f[j+1].requests)):
                f[j], f[j+1] = f[j+1], f[j]
    return f

def test():
    files = order_tape(initialize_tape(20))
    for i in files:
        print("length: ", i.length, "request: ", i.requests, "l/r = ", i.length/i.requests)

test()
```

Output:

```
length:  1 request:  13 l/r =  0.07692307692307693
length:  13 request:  14 l/r =  0.9285714285714286
length:  15 request:  10 l/r =  1.5
length:  18 request:  11 l/r =  1.6363636363636365
length:  15 request:  9 l/r =  1.6666666666666667
length:  26 request:  15 l/r =  1.7333333333333334
length:  27 request:  15 l/r =  1.8
length:  29 request:  9 l/r =  3.2222222222222223
length:  42 request:  11 l/r =  3.8181818181818183
length:  32 request:  8 l/r =  4.0
length:  41 request:  10 l/r =  4.1
length:  17 request:  3 l/r =  5.666666666666667
length:  47 request:  8 l/r =  5.875
length:  16 request:  2 l/r =  8.0
length:  30 request:  3 l/r =  10.0
length:  22 request:  2 l/r =  11.0
length:  12 request:  1 l/r =  12.0
length:  30 request:  2 l/r =  15.0
length:  17 request:  1 l/r =  17.0
length:  27 request:  1 l/r =  27.0
```

Explication:

We create a class a class named files which contains the attributes *length* and *requests*.

We also create a list with some files randomly. In order to do this, we import the randint module from math.

In order to order in the most optimal way, we divide the length by the requests, and we sort the files from minimum to maximum.

In this way, files that are requested many times and have minor length will appear the firsts in the list, which means they will be more accessible and will spend less time. In the other hand, bigger files requested less times will be placed at the end of the list.

EXERCISE 3: Extra Exercise

There is a vector V formed by n data, from which we want to find the minimum element of the vector and the maximum element of the vector. The type of data that is in the vector is not relevant to the problem, but the comparison between two data to see which of them is lower is very expensive, so the algorithm for the search of the minimum and the maximum should make the least amount of comparisons between possible elements.

A trivial method consists of a linear path of the vector to search for the maximum and then another path to search for the minimum, which requires a total of approximately 2n comparisons between data. This method is not fast enough, so it is requested to implement a method with greedy methodology that makes a maximum of $\frac{3}{n}n$ comparisons.

Code:

```python
def minmax(vector):
    max = []
    min = []
    while (len(vector) > 1):
        if (vector[0] > vector[1]):
            max.append(vector[0])
            min.append(vector[1])
            del(vector[0])
            del(vector[0])
        else:
            max.append(vector[1])
            min.append(vector[0])
            del(vector[0])
            del(vector[0])
        if (len(vector) == 1):
            if (vector[0] > max[0]):
                max.append(vector[0])
                del(vector[0])
            else:
                min.append(vector[0])
                del(vector[0])
        maxM = max[0]
        for i in max:
            if (i > maxM):
                maxM = i
        minM = min[0]
        for i in min:
            if (i < minM):
                minM = i
    print("The MIN value is", minM)
    print("The MAX value is", maxM)

vector = [2, 7, 15, 0, 200, 87, 151]
minmax(vector)
```

Output:

```
The MIN value is 0
The MAX value is 200
```

Explanation:

In this exercise, we have to avoid going through the list looking for the minimum and the maximum value, since we would do 2n checks and it would be invalid.

That's why we go through the list taking pairs of values (that is going to allow us to reach the $\frac{3}{n}n$ comparisons), comparing the values to know which one is the maximum (adding it to the list of maximum values) and which one is the minimum (adding it to the list of minimum values). So finally, we would be going through the list doing $\frac{n}{2}$ comparisons.

Then, we would go through both lists looking for the maximum and minimum values. These lists have $\frac{n}{2}$ elements, so going through them would do $\frac{n}{2}$ comparisons, getting that way the final solution with $\frac{3}{n}n$ comparisons.

In addition, when the list is odd and the length of the list is 1, we would check that value with any of the maximum (or minimum) in order to see if is bigger or lower than them. If this condition is not satisfied, it will be added to the opposite list which

EXERCISE 4

We have a non-directed graph G = <N,A>, where N = {1, ..., n} is the set of nodes and A ⊆ NxN is the set of edges. Each edge (i, j) ∈ A has an associated cost cij (cij > 0 ∀i, j ∈ N; if (i, j) ∉ A can be considered cij = + ∞). Let M be the cost matrix of the graph G, that is, M [i, j] = cij (since the non-directed graph is that (i, j) = (j, i) so the matrix M is symmetric). Having as data the number of nodes n and the cost matrix M, it is requested to find the minimum support tree of graph G using Prim's algorithm, using the following ideas:

- Unlike the Kruskal algorithm (which creates the tree using independent connected components that are joined together), Prim's algorithm is based on the idea of building an increasingly large tree, starting with a single node and ending by coating the entire graph.
- The algorithm begins with a tree of a node, to which a second node is added, then a third node, etc., until the n nodes are joined. The way to choose a node is looking for the nearest node to the whole tree, without creating cycles.
- As the size of the tree grows, the search for the nearest node becomes complicated, so for the algorithm to be efficient (the method must have $O(n^2)$) a data structure must be created that stores the best distance from each node to the set of nodes of the tree.
- It will be necessary to store in some way the way in which the tree has been created, for example by indicating to which node of the tree the new selected candidate is joining.

Code:

```python
import math

def PrimAlgorithm(matrix):
    '''
    OBJ: implementation of Prim Algorithm
    '''
    minDistance = [0]
    nearestNode = [0]
    T = []
    knownNodes = []
    for i in range(1, len(matrix)):
        nearestNode.append(0)
        minDistance.append(matrix[i][0])
    for i in range(len(matrix) - 1):
        min = math.inf
        for j in range(1, len(matrix)):
            if (0 <= minDistance[j] < min and j not in knownNodes):
                min = minDistance[j]
                k = j
        T.append([nearestNode[k] + 1, k + 1])
        knownNodes.append(k)
        for j in range(1, len(matrix)):
            if (matrix[j][k] < minDistance[j]):
                minDistance[j] = matrix[j][k]
                nearestNode[j] = k
    return T

w = math.inf
matrix = [[0, w, 4, w, 5, 1, w],
          [w, 0, 3, 6, 2, w, w],
          [4, 3, 0, 5, w, w, 1],
          [w, 6, 5, 0, 9, w, 7],
          [5, 2, w, 9, 0, w, 5],
          [1, w, w, w, w, 0, 3],
          [w, w, 1, 7, 5, 3, 0]]

print(PrimAlgorithm(matrix))
```

Output:

```
[[1, 6], [6, 7], [7, 3], [3, 2], [2, 5], [3, 4]]
```

EXERCISE 6

Shrek, Donkey and Dragona arrive at the foot of Lord Farquaad's towering castle to free Fiona from her confinement. As they suspected that the drawbridge would be guarded by numerous soldiers, many stairs have been brought, of different heights, with the hope that some of them will allow them to overcome the wall; but no stairs serve them because the wall is very high. Shrek realizes that if he could combine all the stairs into one, he would be able to get to the top exactly and be able to enter the castle.

Fortunately, the stairs are made of iron, so with the help of Dragona they go to "weld" them. Dragona can weld any two stairs with her fire breath, but it takes to heat the extremes as many minutes as meters add up the stairs to be welded. For example, welding 6 and 8 meters of stairs would take 6+8=14 minutes. If this ladder was then welded to a 7-meter ladder, the new time would be 14 +7=21 minutes, so it would have taken a total of 14+21=35 minutes to complete the ladder.

Design an efficient algorithm that finds the best cost and way to weld the stairs so that Shrek takes as little time as possible to scale the wall, indicating the chosen data structures and their way of use. It can be assumed that exactly the stairs necessary to climb the wall are available (neither left nor missing), that is, the data of the problem is the collection of measurements of the "mini-ladders" (in the data structure chosen), and that only the optimal way of melting the stairs is looked for.

Code:

```python
def shrek(stairs):
    """
    OBJ: stairs must be welded from minimum to maximum, in order to find the
    optimum solution.
    """
    stairs.sort()
    minimum = 0
    j = stairs[0]
    for i in range(1, len(stairs)):
        if (i == 1):
            j = j + stairs[1]
        else:
            j += stairs[i]
        minimum += j
    return minimum

stairs = [10, 5, 7, 14, 2, 1, 3]
print("The minimum time Dragona needs to weld the stairs is", shrek(stairs))
```

Output:

```
The minimum time Dragona needs to weld the stairs is 108
```

Explanation:

The code is pretty simple. In order to minimize the costs of welding, we must weld from lowest to highest. Doing that, shorter stairs will be summed more times than bigger are. Then, the accumulated sum is minimized.

As a result, the vector containing the stairs is sorted from minimum to maximum. The variable *minimum* accumulates the stairs sum accumulating and it is returned.